Dhrystone and AXD Debug

Kunjian Song

1 Dhrystone Benchmark

1.1 Exploration of Compiler Configuration in ARM Developer Suite:

This section summarizes the observations and findings during self-study. This is the configuration steps used to obtain the results for the Dhrystone table and analysis.

When a new project is created, the source files can be built on three types of targets, *DebugRel*, *Release* and *Debug* shown in Figure 1 below.

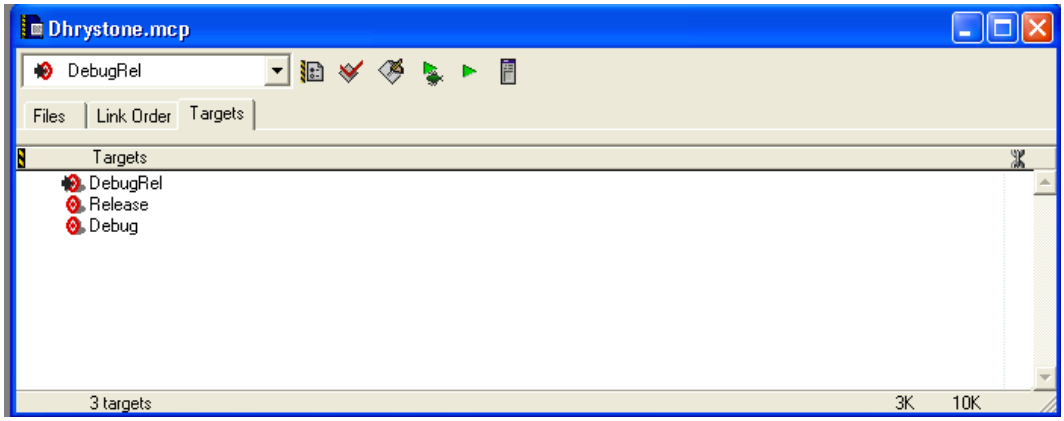

Figure 1: Available 'Targets' selections

Double clicking on the target brings up the Settings window in which the user can change the ARM C compiler configurations, e.g. adding pre-processor flag, enabling the warning and changing the optimization criterion (as shown in Figure 2 below).

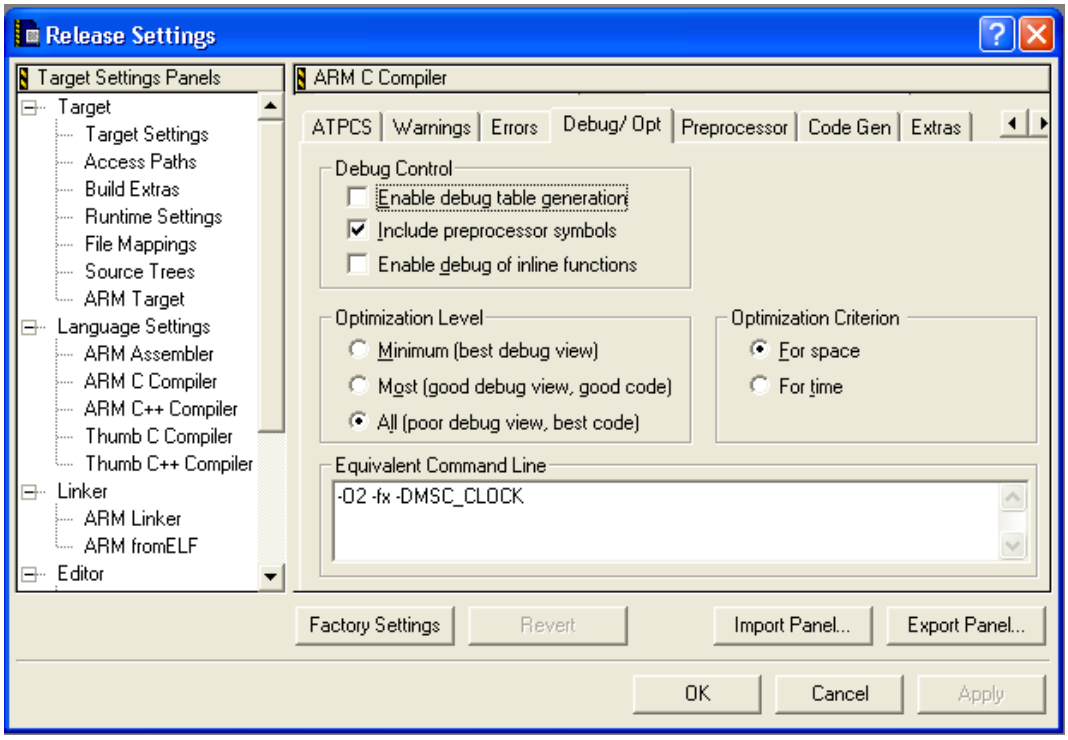

Figure 2: ARM C compiler configuration window for Release Settings where Optimization Criterion can be modified under the Debug/Opt tab.

The MSC_CLOCK needs to be added for each target, otherwise errors occurs due to undefined symbols, e.g. Hz. After adding the MSC_CLOCK, the compiler command line should contain "-DMSC_CLOCK" argument.

Project->Remove Object Code -> All Targets allows the user to rebuild the program when there is nothing changed (in case the *Error & Warnings* window is accidentally closed).

1.2 Dhrystone Table and Analysis

Table 1 summarizes the observations when compiling and running the Dhrystone benchmark. The code size is the value reported by "**Total RAM Size**" in the *Error & Warnings* window.

Project Type	Optimization	Code Size
Debug	Space	29.40kB
	Time	29.40kB
Release	Space	28.79kB
	Time	28.97kB

Table 1: Dhrystone benchmark compilation and run (1000k iterations)

The code size becomes smaller when compiling using "Optimization for Space" option.

It was also found that Release build has an optimization level "**-O2**", but Debug build has an optimization level of "**-O0**".

Different iterations {10k, 50k, 1000k} gives values too small and not computable. Hence the following experiment using iterations of 500k, 1000k, 1500k and 2000k iterations. The following tables summarize the Dhrystone one run through time and Dhrystones per second. The trends are plotted in Figure 3 and Figure 4.

Debug_Space		
Iterations	Dhrystone Run through Time (µs)	#Dhrystones per Second
500k	17.6	56689.3
1000k	20	50125.3
1500k	22.3	44856.5
2000k	22.6	44336.1
Debug_Time		
Iterations	Dhrystone Run through Time (µs)	#Dhrystones per Second
500k	too small	too small
1000k	18.3	54615
1500k	17.8	56179.8
2000k	17.2	58258.1

Table 2: Debug run time - space and time

Release_Space		
Iterations	Dhrystone Run through Time (µs)	#Dhrystones per Second
500k	18.4	54229.9
1000k	17	58651
1500k	16.9	59055.1
2000k	19.9	50150.5
Release_Time		
Iterations	Dhrystone Run through Time (µs)	#Dhrystones per Second
500k	39.1	25562.4
1000k	21.7	46082.9
1500k	25.4	39318.5
2000k	26.8	37369.2

Table 3: Release run time - space and time

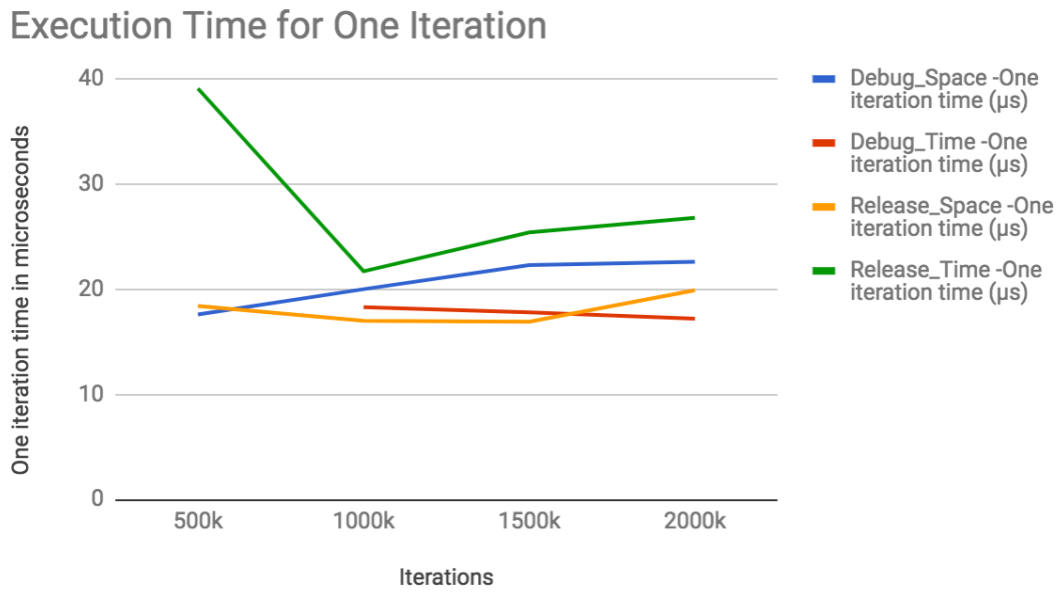

Figure 3: comparison of execution time for one iteration

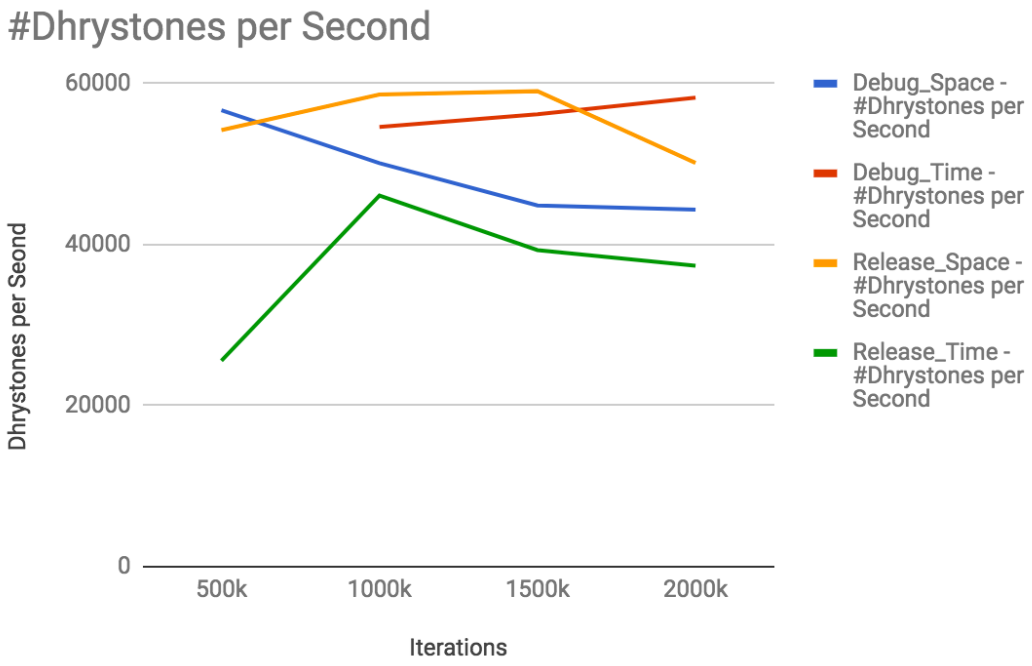

Figure 4: Dhrystones per Second

Based on the above figures, Release_Time has the longest run time and lowest number of Dhrystones per second. Between 1000k to 5000k iterations, Release_Space outperformed others in terms of run time and number of Dhrystones per second.

2 Debugging sort.c Program

2.1 Basic Debugging using Watchpoint

This section describes the findings and new techniques learnt in the basic debugging exercise. This section only summarizes the usage of watchpoint, as the usage of

breakpoints in AXD is fairly similar to other debuggers (e.g. GDB). The following figures show how to set watchpoint of a variable for future reference. **These steps and techniques are also used for the next section to debug sort.c program**.

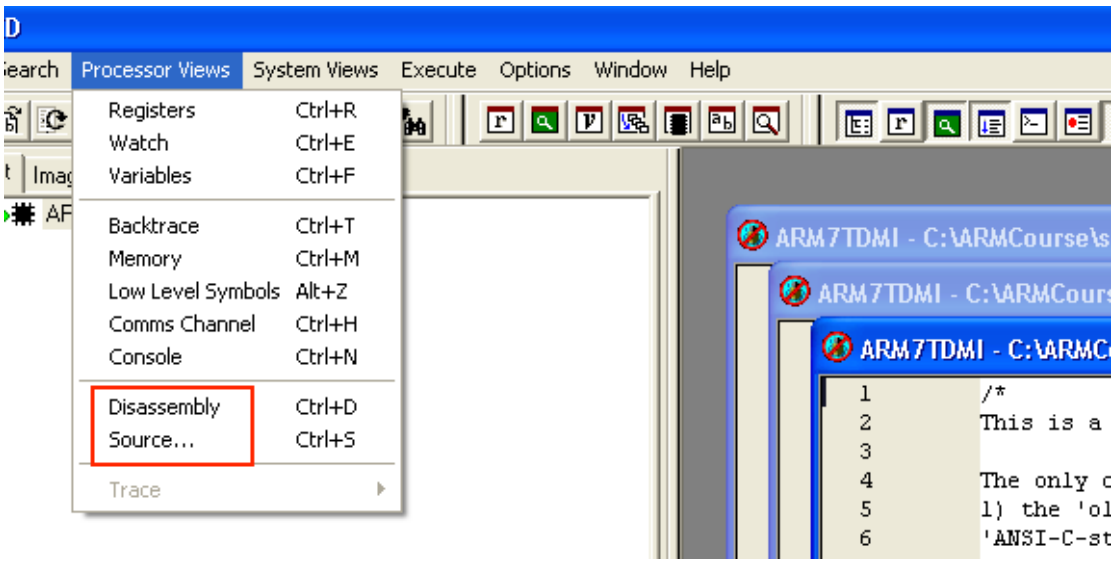

Figure 5: Use Processor Views menu to look into source or disassembly code

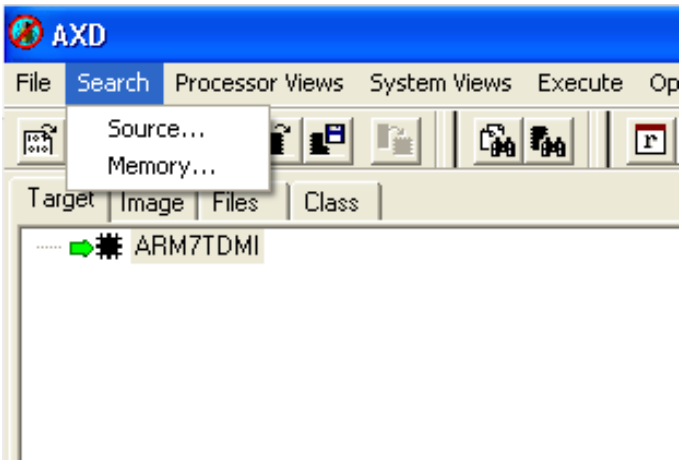

Figure 6: Use Search menu to search a variable name or function name in the source code.

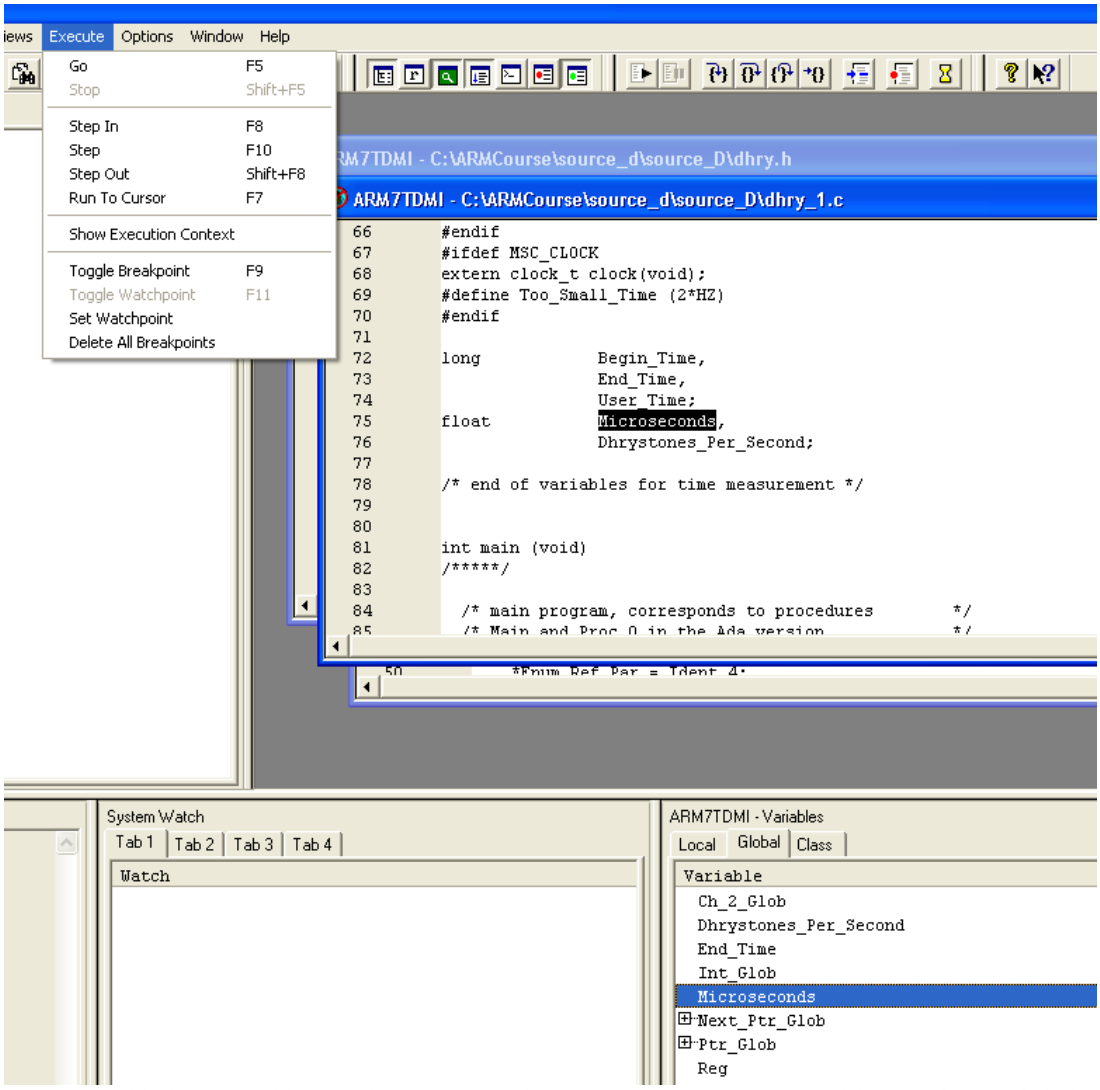

Figure 7: Highlight a variable in either source code or Processor Views->Variables window and click on Execute->Set Watchpoint to set the watchpoint for that variable

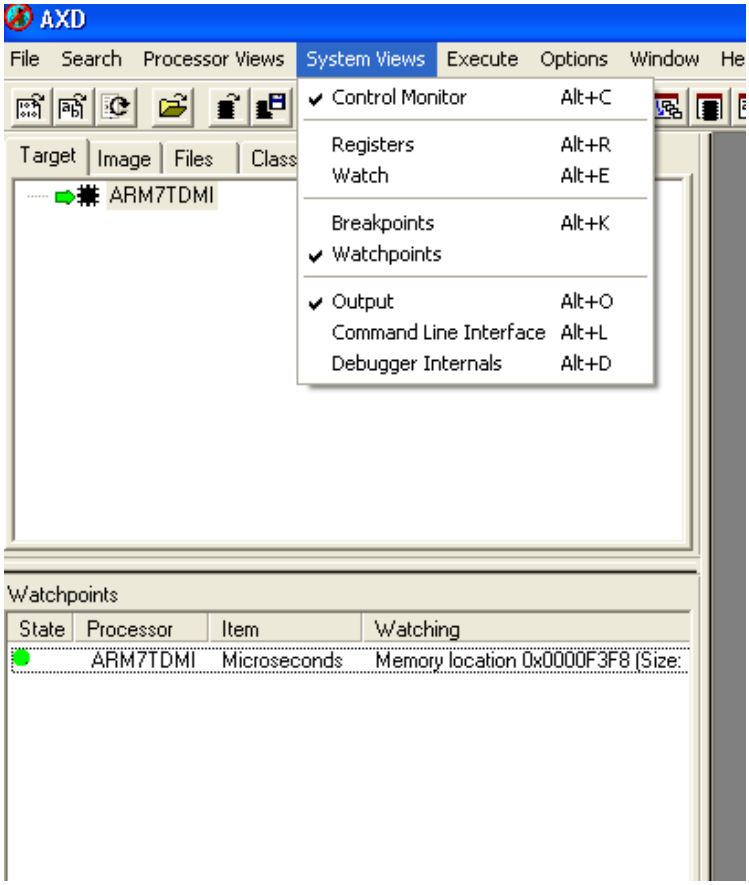

Figure 8: Select System vViews -> Watchpoints to confirm that the variable is being watched!

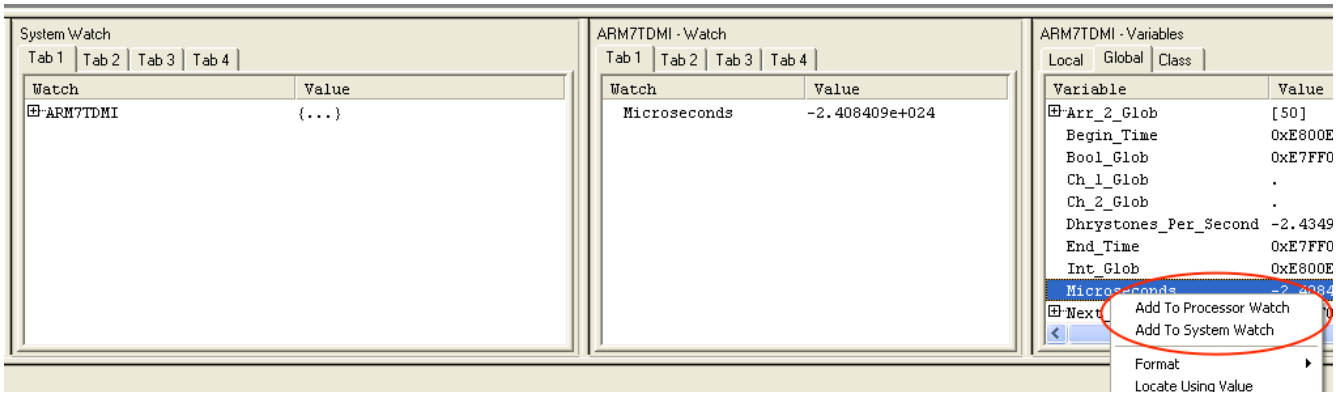

Figure 9: right click on the variable and choose "Add to Process Watch" and "Add to System Watch" to view the change of that variable in System Watch and Processor Watch window

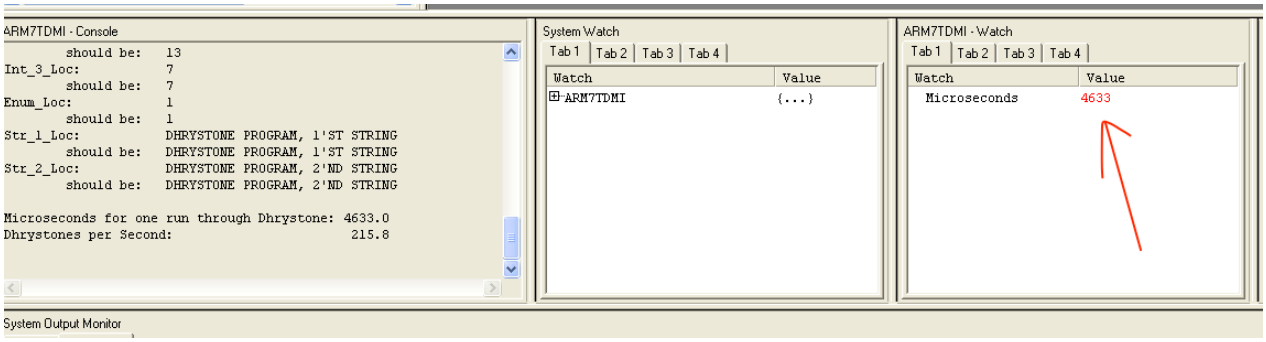

Figure 10: When the variable being watched is changed, the colour becomes red

2.2 Bug in sort.c Program

The sort.c program consists of the code blocks for the following sorting algorithms:

- Insertion sort for problem size <= 1000
- Shell sort for problem size > 1000
- Quick sort for problem size > 1000

Depending on the problem size, different block of code will be compiled as defined by the "#if" compiler directive.

As N is defined as integer 1000, insertion sort algorithm will be compiled and used. The original error message was "Insertion sort failed - exiting" which comes from *check_order* function. The program failed at line 114 when calling *check_order* function after sorting the string by insertion sort. Hence the first step is to add a breakpoint at Line 114 and step into that function to check why it failed.

To check the values of the local variable when the bug is triggered during run time, inside the *check_order* function, another breakpoint is set at line 81 inside the *if* construct as shown in the Figure 11 below.

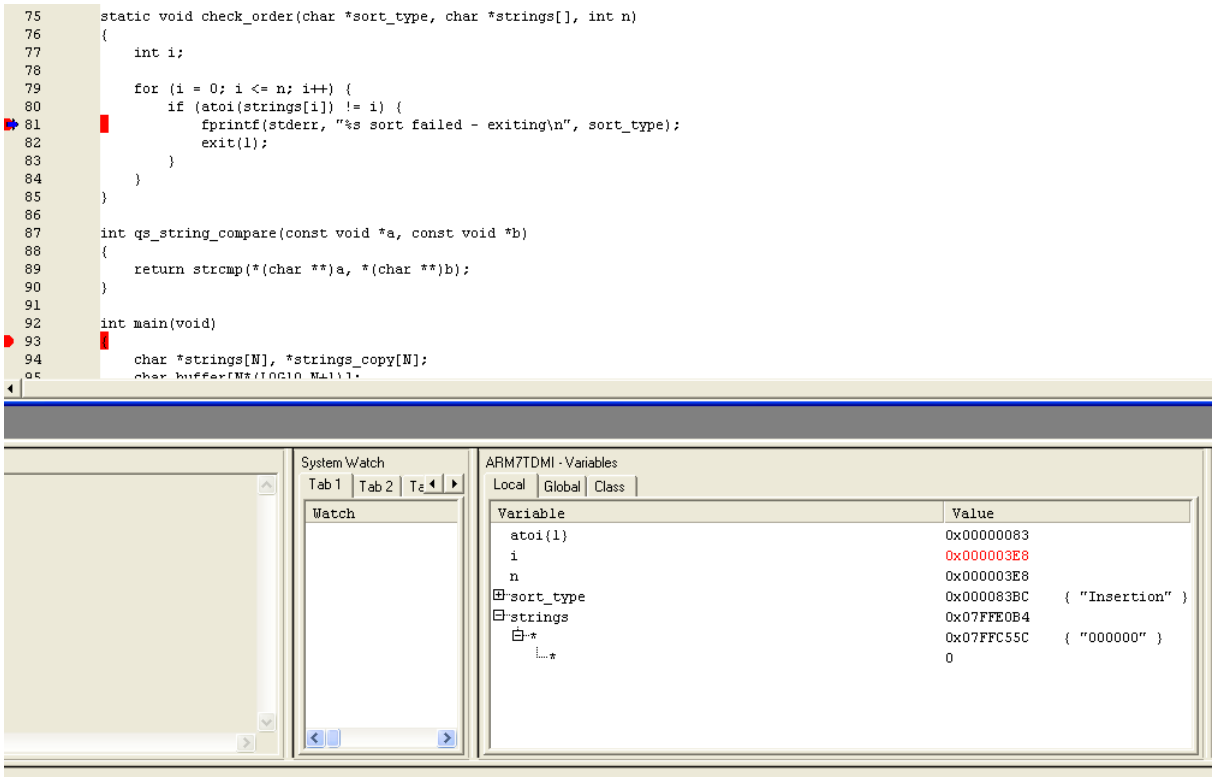

Figure 11: When bug is triggered

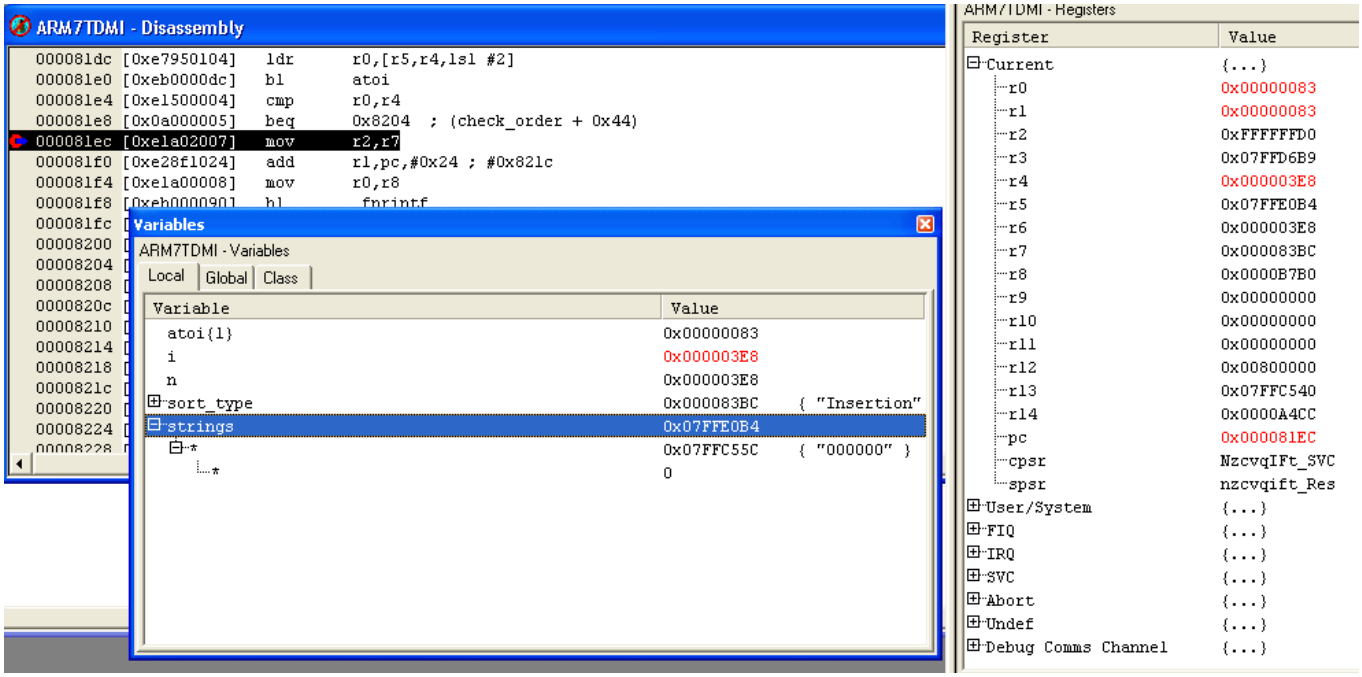

Figure 12: Assembly code, register values and local variables.

From the code and local variable values, we can see that this is a common C array out-of-boundary problem - “<” should be used instead of “<=”.